Multi-Target C++ Implementation of Parallel Skeletons

Wilfried Kirschenmann
EDF R&D & AlGorille INRIA project team
1 av. du Gal de Gaulle
F-92140 Clamart, France
wilfried.kirschenmann@edf.fr

Laurent Plagne
EDF R&D
1 av. du Gal de Gaulle
F-92140 Clamart, France
Laurent.plagne@edf.fr

Stephane Vialle
SUPELEC - IMS group & AlGorille INRIA project team
2, rue Edouard Belin
F-57070 Metz Cedex, France
stephane.vialle@supelec.fr

ABSTRACT

This paper presents the design of an efficient multi-target (CPU+GPU) implementation for the `Parallel_for` skeleton. Emerging massively parallel architectures promise very high performances for a low cost. However, these architectures change faster than ever. Thus, optimization of codes becomes a very complex and time consumming task. We have identified the data storage as the main difference between the CPU and the GPU implementation of a code. We introduce an abstract data layout in order to adapt the data storage. Based on this layout, the utilization of `Parallel_for` skeleton allows to compile and execute the same program both on CPU and on GPU. Once compiled, the program runs close to the hardware limits.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Parallel programming*

General Terms

C++ templates, parallel computing, Nvidia CUDA, Intel TBB, parallel skeletons, data layout

1. MOTIVATION AND MAIN OBJECTIVES

In many scientific applications, computation time is a strong constraint. Optimizing for the rapidly changing computer hardware is a very expensive and time consuming task. Emerging hybrid architectures tend to make this process even more complex.

The classical way to ease this optimization process is to build applications on top of High Performance Computing (HPC) libraries. Each HPC library allows the scientific developer to use a well defined Application Programming Interface (API) tailored for its specific scientific sub-domain. Because of their limited scope, it is possible to produce specialized HPC implementations of these libraries for a large variety of hardware target architectures. Hence, scientific applications for which the computing time is mostly consumed within HPC library subroutines, automatically exhibit optimal performances for various hardware target architectures.

However, HPC libraries like MKL [8] or ATLAS [23] implement classical APIs like BLAS [14] that may be too rigid to match the needs of potential client scientific applications. Following the model of the C++ Standard Template Library (STL) [17], template based *generic libraries* such as Blitz++ [22] provide much more flexible interfaces and thus extend the potential use of library based design for scientific applications. As an example the Matrix Template Library (MTL4) [12] allows to deal with mixed precision Linear Algebra (LA) operations that are outside the scope of the rigid BLAS API. Moreover, some generic libraries allow the definition of Domain Specific Embedded Languages (DSELs) [6]. For example the STL provides containers and algorithms which can be combined to create new concepts.

Legolas++ is a generic library developed at EDF R&D that provides building blocks for the specific domain of Highly Structured Sparse Linear Algebra (HSSLA) problems arising in many simulation codes. In particular, it allows to deal with recursively blocked matrices (matrix of blocks of blocks of...) that appear for example in neutron transport simulations [16]. In order to build HPC codes meeting EDF's industrial quality standards, a *multi-target* version of Legolas++ is presently developed that should provide a unified interface for both multi-core CPUs and Graphics Processing Units (GPUs) optimal implementations.

Not all but a large fraction of the Legolas++ operations are embarrassingly parallel and consist in applying the same function independently on multiple data. This kind of problems are well described with a `Parallel_for` algorithm that is an instance of parallel algorithm skeletons introduced in [4]. In this article we propose a design for a C++ multi-target (CPU/GPU) implementation of the `Parallel_for` skeleton. This implementation will be named MTPS after the title of this paper.

This paper is structured as follows. In Section 2, we introduce the problem class that can be handled by our implementation. Section 3 describes how MTPS is to be used to define the application presented in Section 4. Performances obtained are discussed in Section 5. Section 6 compares our works with other approaches. Finally, the conclusion follows in Section 7.

2. CONSIDERED CLASS OF PROBLEMS

The execution time of a given code depends both on the

POOSC '09, July 7 2009, Genova, Italy

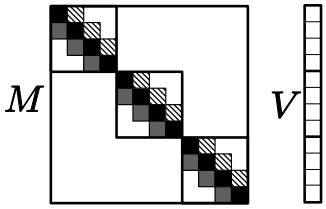

Figure 1: 3×3 block-diagonal matrix with 4×4 tridiagonal symmetric blocks (left) and a 3×4 vector (right).

time required to make the computations and on the time required to load and store data from and to the main memory. During the last years, processors performances have increased a lot faster than the memory bandwidth [24, 2]. As a consequence of this evolution, the limiting factor of solving a given problem often appears to be the memory access speed. As explained in Section 3.1, the memory access pattern which optimizes the data throughput differs from CPU to GPU. Memory access patterns depend on both the data storage layout and the execution data flow. In order to adapt the memory access patterns to the target architecture, we propose a specific data structure that abstracts the actual data storage layout and we implement a target-specialized parallel skeleton driving the execution data flow. In order to delimit rigorously the scope of application of our approach, this section presents a formal definition of the problems that can be handled. This abstract definition will be illustrated by one example of application introduced in the following paragraph.

2.1 An example of application

In this Section we introduce one application composed of two operations. This application will be used to illustrate the abstract definitions.

Let us consider a block-diagonal matrix M whose blocks are TriDiagonal Symmetric Matrices (TDSM), positive definite, and a block vector V. Figure 1 shows a 3×3 block diagonal matrix with 4×4 TSDM blocks and a block-vector containing 3 block of 4 elements. Similar matrices appear in many physics problem discretized with a cartesian spatial mesh (e.g. particle transport [16]).

In order to compute X such as $MX = V$ (or $X = V/M$) where M is symmetric, positive definite, one common solution is to factorize M using a Cholesky decomposition $M = LL^t$ where L is a lower triangular matrix. This factorization allows to solve the linear problem with a *forward* substitution ($LX' = V$) followed by a *backward* substitution ($L^tX = X'$). The interrested reader may refer to the book by Golub & Van Loan [7] for further details. We have chosen an in-place implementation. During the factorization, M is replaced by its factorized form. The same way, after the xomputation, V is updated to contain V/M.

Both the `factorize` and the `substitute` operations can, be applied *independently* on each block of the matrix. Let us define a *collection* as a set of independent operands for a given operation. Using this terminology, M and V can be described as collections of TDSM blocks (M_{block}) and of vector blocks (V_{block}) with respect to the `factorizeBlock` and the `substituteBlock` operations as shown in Algorithm 1 and 2. Algorithms corresponding to these two functions are provided in Section 4.

Algorithm 1: factorize

Data: A block-diagonal matrix M
Result: M is factorized
forall $M_{block} \in M$ **do**
  `factorizeBlock`(M_{block})

Algorithm 2: substitute

Data: A factorized block-diagonal matrix M
Data: A block vector V
Result: $V = V/M$
forall $(M_{block}, V_{block}) \in (M,\ V)$ **do**
  `substituteBlock`($(M_{\text{block}}, V_{\text{block}})$)

2.2 Abstraction and Generalization

Based on the previous application, the following definitions provide an abstraction and a generalization of the problems that can be written using our approach.

2.2.1 Data Structure

Matrices and vectors shown on Figure 1 have comparable data structures: both are collections of objects (blocks) whose data elements can be accessed using 1 (vector) or 2 (matrices) indexes. This section precises the data structure and the constraints on it.

- Def.: a *vector* is a set of distinct elements of the same type and identified by an integer index.
- Let $\mathbb{V}$ be the set of classes which allow to define *vectors*. Let `V<T>` $\in \mathbb{V}$ be a class of *vector* of `T`. We assume that the `i`$^{\text{th}}$ element of an instance `v` of `V<T>` can be accessed by `v[i]` and that the number of elements can be obtained by `v.size()`. Note that this definition does not precise the underlying storage of the class `V<T>`. For instance, the diagonal and the lower-diagonal of the matrix blocks are both of type `V<float>`.
- Let $\mathbb{O}$ be the set of classes whose attributes *can* be written as a unique *vector* of *vector* of single precision floating point numbers. Take for example `O` a class element of $\mathbb{O}$:

```
1 struct O {
2   V< V<float> >  fields_;
3 };
```

  For instance, in TDSM blocks, `fields_.size()` is 2 and `fields_[0]` corresponds to the vector containing all diagonal elements while `fields_[1]` corresponds to the vector containing all lower-diagonal elements.

  Some example of classes in $\mathbb{O}$:

Mathematical objects	Number of vectors	Size of vectors
Vector of size N	1	N
Matrix of size $N \times N$	N	$N, \ldots, N$
Tridiagonal matrix of size $N \times N$	3	$N-1$, N, $N-1$
TDSM of size $N \times N$	2	N, $N-1$

- Let $\mathbb{V} < \mathbb{O} >$ be the set of *vectors* whose elements are instances of a class in $\mathbb{O}$.

Example : Let $\texttt{V} < \texttt{O} > \in \mathbb{V} < \mathbb{O} >$ be a *vector* of element of type `O`. Let `v` be an instance of $\texttt{V} < \texttt{O} >$ and containing $N_{\texttt{elts}}$. Each element of the *vector* can be accessed using its index:

$$\forall \texttt{i} \in [0, N_{\texttt{elts}} - 1] \qquad \texttt{v[i]}$$

The matrix shown in Figure 1 is a *vector* of 3 TDSM blocks denoted by $M[0]$, $M[1]$ and $M[2]$.

2.2.2 *Parallel_for Skeleton*

In order to be used in our parallel skeletons, some constraints concerning the operations using the previously defined data structure have to be respected.

- Def.: a *n-tuple* is an ordered set containing n elements not necessarily distinct. A *n-tuple* is noted using braces () (e.g.: (a_n) is a *n-tuple* of $a_0 \dots a_{n-1}$).
- Def.: a *collection* c_f is a vector of object instances of a class $\texttt{O} \in \mathbb{O}$ whose elements are independent with respect to a given function f. All elements of a *collection* must have the same number of fields (they are instances of the same class) which have the same size.
- Let $\mathcal{A}$ be the set of all n-tuples $(o_0, o_1, \dots, o_n)$ of object instances of classes $(\texttt{O}_0, \texttt{O}_1, \dots, \texttt{O}_n) \in \mathbb{O}^n$.
- Let f be a function taking as argument an object a in $\mathcal{A}$. Let c_f be a *collection* of $N_{\texttt{elts}}$ elements of $\mathcal{A}$ for f: all elements of c can be passed **independently** as arguments to f. Let F be the function that applies f on all elements of c:

$$F(c_f) \equiv f(c_f[i]) \quad \forall i \in [0, N_{\texttt{elts}} - 1]$$

- Let $\mathcal{F}$ be the set of all functions F that apply a function f independently on all elements of a collection c_f

Considering the forward and backward substitution presented in Algorithm 2, the arguments of `substituteBlock` are $(M_{\text{Block}}, V_{\text{Block}})$. As M_{Block} and V_{Block} are objects whose classes are in $\mathbb{O}$, the 2-tuple $(M_{\text{Block}}, V_{\text{Block}})$ is in $\mathcal{A}$. Finally, the collection of arguments $c_{\texttt{substituteBlock}}$ is (M, V) with $c_{\texttt{substituteBlock}}[i] = (M[i], V[i])$.

Figure 2 shows the corresponding data structure:

- each dark grey box corresponds to a data *vector*,
- a *vector* of 3 matrix blocks corresponds to the matrix M. Each matrix block ${}_iM_{block}$ contains two data *vectors* corresponding respectively to the diagonal and the lower diagonal,
- a *vector* of 3 vector blocks corresponds to the vector V. Each vector block ${}_iV_{block}$ contains one data *vectors*,
- the dashed box show the 2-tuples of arguments that are passed to `substituteBlock`. The argument collection for `substituteBlock` represented by the dark-gray box.

Previous definitions implies that when a functions f manipulates an object of the n-tuple $c[i]$, only objects in this n-tuple are manipulated. Combined with the proposed data structure, this constraint allows to control the memory access pattern precisely. This way, the optimizations concerning the data structure can be hidden to the user. The implementation of MTPS presented in Section 3 allows a parallel and multi-target processing (CPU+GPU) of operations in $\mathcal{F}$ using a `Parallel_for` skeleton:

$$F(c_f) = \texttt{Parallel\_for}(f, c_f).$$

2.2.3 *Extension*

Using the previous definitions, objects containing other data than of type `float` are not defined. We extend previous definitions to allow objects containing data of different types.

- Let $\mathbb{I}$ be the set of *arithmetic, built-in types* as defined in Chapter 4.1.1 of Stroustrup's *C++ Programming Language* [19]:
  $\mathbb{I} = \{\texttt{float}, \texttt{double}, \texttt{int}, \texttt{bool}, \texttt{long}, \texttt{char}, ...\}$
- Let $\mathbb{M}$ be the set of classes whose attributes *can* be written as *vector* of *vector* of elements of $\mathbb{I}$. Take for example `M` a class element of $\mathbb{M}$:

```
1 struct M{
2   V< V<float> >  floatFields_;
3   V< V<double> > doubleFields_;
4   V< V<int> >    intFields_;
5   ...
6   // one vector for each
7   //arithmetic, built-in type
8 };
```

The proposed implementation will be generalized to collections of objects $\in \mathbb{M}$ whose fields are extended to all types of $\mathbb{I}$.

3. IMPLEMENTATION OF THE PROBLEM

During the parallelization of a physics solver on GPU [11], several embarrassingly parallel operations in $\mathcal{F}$ have been implemented. To ease further developments, we aim at writing a multi-target DSEL for HSSLA: Legolas++. This high level DSEL will rely on a lower level DSEL dedicated to data parallelism. MTPS, whose implementation is presented here, is a first step to the design of this lower level data parallel DSEL. Using the MTPS DSEL, a user must:

- specify the target machine (`DEVICE` in the following);
- describe the data structure of the blocks;
- build a collection of blocks;
- write a function to be applied to all blocks of a collection;
- apply the function in parallel using parallel skeletons.

Next, we show how to describe the data structure, then we focus on the data manipulation. At last, a full example of utilization will be given.

3.1 Polymorphic Data Layout

As shown in Figure 3, the optimal memory access pattern is not the same on CPUs as on GPUs. On CPUs (left), a contiguous range of data must be accessed by a thread in order to fit in the cache memory. In this example, all dark grey data are loaded in the dark grey cache memory during the first loop iteration. In following iterations, data are already in the cache memory and accesses are very fast. On a GPU (right), two contiguous *threads* must access two

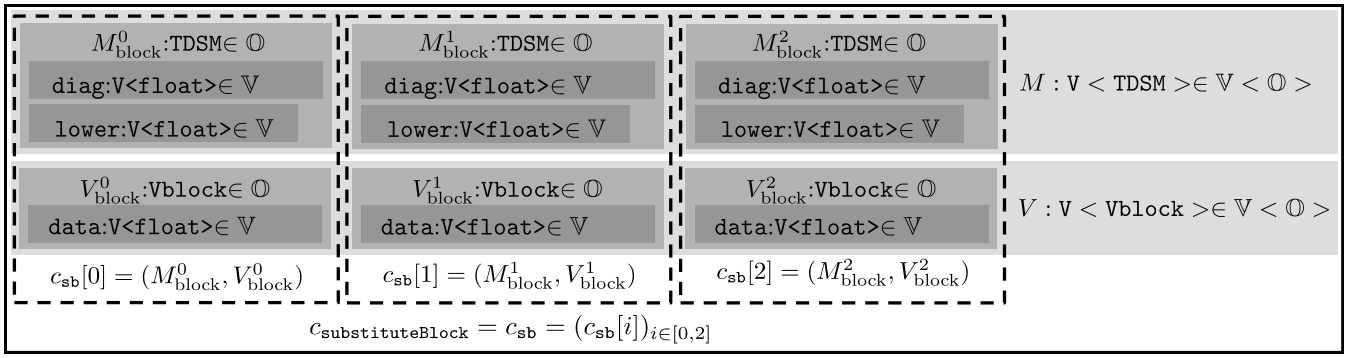

Figure 2: Data structure corresponding to the arguments of the forward and backward substitution: a 3×3 block-diagonal matrix and a vector with 3 blocks as shown on Figure 1. The notation `instance : Class` $\in$ `ClassSet` is to be read as: `instance` is of type `Class` which belongs to `ClassSet`.

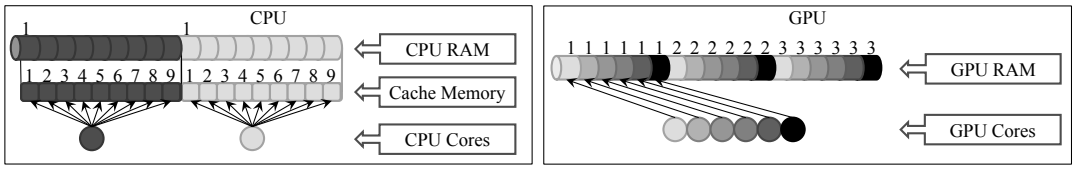

Figure 3: Differences between memory access patterns on CPU and on GPU. In this example, one thread is running on each core. On GPU, we use NVIDIA's definition [15] of cores: a core is a processing unit of a SIMD multiprocessor.

contiguous data items to minimize the total number of access instructions. In this example, each each data is loaded on the core withe the same color. first, the 6 data numbered "1" are loaded synchronously during one access, then the data numbered "2" and finally the data numbered "3". Compared to the CPU, the CPU cores compute each 3 data element that are interlaced with data computed by the 5 other cores.

In the following of this Section, we note c_f a collection of objects $c_f[i]$ on which f will be applied in parallel.

On a CPU, efficient utilization of the cache memory requires each thread data in a contiguous memory space. In other words, the two following elements have to be stored contiguously[1]:

- `v[i].fields_[j][k]`
- `v[i].fields_[j][k + 1]`.

On a GPU, efficient utilization of the memory controler requires two contiguous threads to access two contiguous elements. Therefore, in each field, data have to be interlaced. If we consider that a contiguous range of threads will compute contiguous ranges of data as shown in Figure 3 (i.e. thread j computes $f(c_f[i])$ and thread $j+1$ computes $f(c_f[i+1])$), then the two following elements must be stored contiguously:

- `v[i].fields_[j][k]`
- `v[i + 1].fields_[j][k]`.

Figure 4 shows how the data of the matrix M and the vector V are effectively stored. In both cases, the different fields can be stored separately as there is no constraint concerning their continuity. Thank to this, building a collection of n-tuple is not mandatory. Instead, we provide vector of objects. If an operation requires several objects from different vectors, this operation will work on several vectors and build the n-tuples of objects internally.

As a consequence, memory allocation and data initialization must be done at the level of vectors of objects. Thus the definition of classes in $\mathbb{O}$ must be independent of the storage. To ease users' developments, we have chosen a descriptive approach for classes in $\mathbb{O}$. In the MTPS design, the user must write a class that defines:

- the data type;
- the number of fields;
- a method to compute the size of each field (e.g.: size of a vector);
- a name for each of the fields.

This class must inherit from:
`MTPS::FlatObject<RealType, nbFields>`.

The following user-derived class shows the definition of the `TDSMDef` class which describes the structure of the TDSM blocks of Section 2.1. As its elements are stored in two fields, `TDSMDef<Realtype>` inherits from `MTPS::FlatObject<RealType, 2>`. The first field corresponds to the diagonal while the second corresponds to the lower diagonal. Section 4 describes an application using such matrices.

```
1 template <typename T>
2 class TDSMDef
3   : public MTPS::FlatObject<T,2>{
4   private:
5     typedef MTPS::FlatObject<T,2> Fo;
```

[1]This is not exactly true when using SSE units. However, in a first time, we rely on compilers to use SSE units

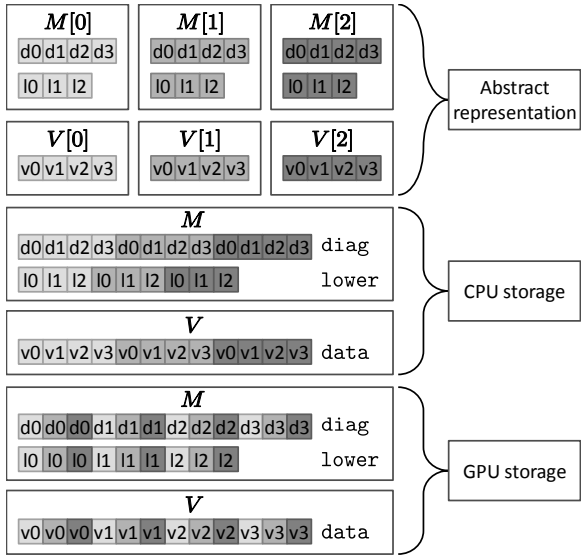

Figure 4: MTPS polymorphic data layout. This figure show how the data storage is adapted to optimize the access pattern. Each color corresponds to an index in the vector.

```
6   public:
7     typedef typename Fo::Shape Shape;
8
9     // MTPS data structure description
10    enum Fields{
11      diag = 0,
12      lower = 1
13    };
14
15    // Method the size of the fields
16    static Shape createShape(int size){
17      Shape shape;
18      shape[diag]=size;
19      shape[lower]=size-1;
20      return shape;
21    }
22
23    // Other user-defined functions
24    static int getSize(const Shape & s){
25      return s[diag];
26    }
27    ...
28 };
```

This code shows the following elements:

- the `Fields` enum (lines 9 to 13) gives the name of each field,
- the static function `createShape` (lines 15 to 21) computes the size of each field,
- the static function `getSize` (lines 21 to 23) is required by the user's application and not by the model.

With this definition, it is possible to create collections whose memory representations are automatically specialized for the target device defined by the type `Device` corresponding either to the CPU (TBBDevice) or the GPU (CUDADevice):

```
1 typedef TDSMDef<float> TDSM;
2 typedef MTPS::Vector<TDSM, Device> TDSMVec;
3 TDSM::Shape ms = TDSM::createShape(
      blockSize);
4 TDSMVec tdsmVec(ms, nbBlocs);
```

The `blockSize` and `nbBlocks` variables correspond to the size of matrix blocks and their number as defined in Section 4 and in Figure 1.

3.2 Operation on Collections

Now that tools to create vectors in $\mathbb{V} < \mathbb{O} >$ have been introduced, this section will show how to define a functor to manipulate them. To understand the proposed API, readers must be aware of the following:

- The devices do not necessarily share the same memory space. For instance, a memory address in GPU memory is not accessible from the CPU. This implies that the different devices do not offer the same features. For instance, initializing GPU data from files on a hard disk is not directly possible. One must first read the file (using the CPU) and then copy the informations to the GPU memory on the GPU card. In order to hide these copies, the concept of *Views* has been introduced. Vectors in $\mathbb{V} < \mathbb{O} >$ can only be modified through function F that can see them as collections. Therefore only Views on collections (`ColViews`) are provided. According to the way a functor will manipulate a collection, three different `ColViews` can be used:
  - `ColOutView<>` for collections that will be written before being read. This View is typically used in initializations,
  - `ColInView<>` for collections that will only be read only,
  - `ColInOutView<>` for collections that will be read and written.

  These views are constructed from a collection. They define:
  - `operator[i]` returning the $\mathtt{i}^{\text{th}}$ element in the collection,
  - `nbElts()` returning the size of the collection.
- Manipulated objects are always part of a collection and do not exist on their own. This implies that only views of objects (`ObjViews`) can be created. `ObjViews` provide two kinds of operators to access data. Let `view` be a view on an element $c_f[i]$:
  - `view(field,k)` $\rightarrow$ $c_f[i]$.`fields_`[`field`][`k`],
  - `view[k]` $\rightarrow$ $c_f[i]$.`fields_`[0][`k`].

  The latter has been introduced to simplify the utilization of objects containing only one field such as our block vector V.

As the `ObjectViews` inherit from their `ObjectDefinition`, all functions defined by the user in `ObjectDefinition` are available in `ObjectViews`. Now, we will introduce `TDSMInit`, a class functor that is used to initialize a `TDSMCol`. The diagonal elements are initialized to `dValue` while elements on the lower

diagonal are initialize to `lValue`. In this functor, the constructor builds the `ColOutView` on the TDSM collection (line 20). Then views on each element are built in line 25. Note on line 27 that this functor uses the user-defined function `getSize`. This function, provided in the `TDSMDef` class is inherited in the class `MView`.

```
1  template <typename TDSM_COL,
2            DEVICE=...,
3            ...>
4  class TDSMInit{
5    public:
6      typedef DEVICE Device;
7    private:
8      typedef TDSM_COL MC;
9      typedef MTPS::ColOutView<MC> MCView;
10     typedef typename MCView::ObjView MView;
11
12     MCView mcView_;
13     float  dValue_;
14     float  lValue_;
15
16   public:
17     TDSMInit(MCollection & mc,
18              float         dValue,
19              float         lValue)
20       : mcView_(mc),
21         dValue_(dValue),
22         lValue_(lValue){}
23
24     void operator() (int i) const {
25       MView m = mcView_[i];
26       const int size =
27         MView::getSize(m.getShape());
28       for (int i = 0 ; i<size-1 ; i++){
29         m(MView::diag , i) = dValue_;
30         m(MView::lower, i) = lValue_;
31       }
32       const int i = size-1;
33       m(MView::diag, i) = diag_;
34     }
35     int getNbElement() const {
36       return mcView_.nbElts();}
37 };
```

3.3 Processing Collections: Parallel_for

`MTPS::Parallel_for` is a generic function that applies an operation f to all elements of a collection of arguments (c_f). This operation and (c_f) must be encapsulated in a functor. Section 3.2 provides `TDSMInit` as an example of a functor that can be passed to `MTPS::Parallel_for` .

```
1  //  FUNCTOR must provide:
2  //  - void operator()(int i) const
3  //       that modifies only the ith element
4  //  - int getNbElement() const
5  //       that gives the number of elements
6  //  - typedef Device
7  template <typename FUNCTOR>
8  void MTPS::parallel_for(const FUNCTOR & f);
```

Finally, after having defined the elements of the collections and the functors operating on them, users can write a program as:

```
1  int main(){
2    typedef MTPS::CUDADevice Device;
3    //typedef MTPS::TBBDevice Device;
4
5    typedef TDSMDef<float> TDSM;
6    typedef MTPS::Vector<TDSM, Device>
         TDSMVec;
7    const unsigned BlockSize   = 100,
8                   nbBlocks    = 30;
9    TDSM::Shape    matrixShape =
10       TDSM::createShape(BlockSize);
11   TDSMVec tdsmVec(matrixShape, nbBlocks);
12
13   typedef TDSMInit<TDSMCol> InitFunctor;
14   InitFunctor initFunctor(tdsmVec, 2, 1);
15   MTPS::parallel_for(initFunctor);
16 }
```

This program may be divided into three parts:

- the choice of target device (lines 2 and 3). In the current implementation of MTPS, a user can use either `MTPS::CUDADevice` to execute the skeletons on GPU using CUDA [15] or `MTPS::TBBDevice` to execute them on CPU using Intel's TBB [18],
- the construction of collections (lines 5 to 11),
- the construction and parallel execution of the functors (lines 13 to 15).

As we will show in the next Section, the adaptative data layout used allow to write one generic code that runs efficiently either on CPU or on GPU. According to the target device, several optimizations are made, including memory access pattern adaptation.

Figure 5 shows which part of the application have to be developed by the user and what is done by the library.

4. APPLICATION

In the previous section, the interfaces to describe the operations and their data have been introduced. Their utilization to execute the program either on CPU or on GPU has been presented on a very simple example: `TDSMInit`. In this Section two other examples will be shown. The first example is the Cholesky decomposition of a block-diagonal matrix M with TDSM blocks. The second example uses the previously factorized matrix M to solve a linear system $V = V/M$ in-place. This is done using a forward and backward substitution. Figure 1 shows a 3×3 block diagonal matrix with 4×4 TDSM blocks. Similar matrices appear in many physics problem discretized on Cartesian spatial meshes (e.g.: particle transport [16]). Such a matrix may be considered as a collection of `TDSMDef` as defined in Section 3.1.

4.1 Cholesky Decomposition

In this implementation, diagonals and sub-diagonals are stored in two fields $D =$ `diag` and $L =$ `lower`. D_i (respectively L_i) is the i^{th} diagonal (respectively sub-diagonal) element. To avoid divisions in the forward and backward substitution, the factorized matrix contains the inverse of the diagonal values. The in-place decomposition algorithm is:

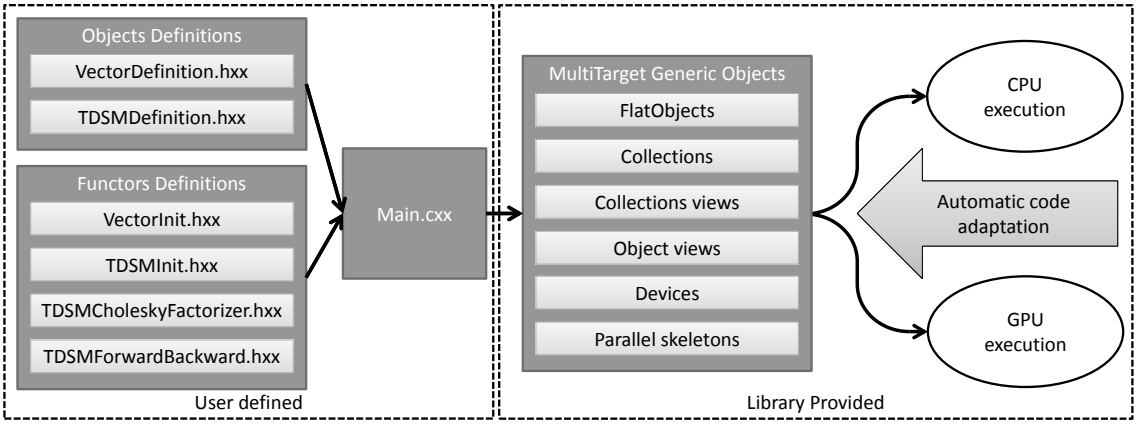

Figure 5: Overview of the relation between client code (left) and library (right). The user-defined "main" program will be statically and automatically adapted to CPU or GPU by the compiler due to the usage of the Multi-target Generic Objects.

Algorithm 3: Cholesky Decomposition of a TDSM

$D_0 \leftarrow (D_0)^{-\frac{1}{2}}$
$L_i \leftarrow D_0 \times L_i$
for $i \leftarrow 1$ **to** $matrixSize - 2$ **do**
  $D_i \leftarrow \left(D_i - (L_{i-1})^2\right)^{-\frac{1}{2}}$
  $L_i \leftarrow D_i \times L_i$
$D_{\text{matrixSize}-1} \leftarrow \left(D_{\text{matrixSize}-1} - (L_{\text{matrixSize}-2})^2\right)^{-\frac{1}{2}}$

This algorithm is then implemented as follows in the class functor `TDSMCholeskyFactorizer`, in the file *TDSMCholeskyFactorizer.hxx* shown in Figure 5:

```
1  void operator() (int i) const {
2    MView m = mcView_[i];
3    const int size =
4      MView::getSize(m.getShape());
5    float lower_=0;
6    for (int j=0 ; j<size-2 ; j++){
7      float diag_=1/sqrtf(m(M::diag, j)-
8                          lower_*lower_);
9      m(MView::diag, j)=diag_;
10     lower_=diag_ * m(M::lower, j);
11     m(M::lower, j)=lower_;
12   }
13   m(M::diag, size-1)=1/sqrtf(
14     m(M::diag, size-1)-
15     lower_ * lower_);
16 }
```

The variable `lower_` is used to reduce the number of memory accesses: it stores `m(M::lower, j)` between two iterations

4.2 Forward and Backward Substitution

Now that the matrices have been factorized, it is possible to solve a linear system such as $V = V/M$ using a forward and backward substitution. The algorithm reads as:

Algorithm 4: Forward and Backward Substitution on a Factorized TDSM

$V_0 \leftarrow V_0 * D_0$
for $i \leftarrow 1$ **to** $matrixSize - 1$ **do**
  $V_i \leftarrow (V_i - V_{i-1} * L_{i-1}) * D_i$
$V_{\text{matrixSize}-1} \leftarrow V_{\text{matrixSize}-1} * D_{\text{matrixSize}-1}$
for $i \leftarrow 1$ **to** $matrixSize - 1$ **do**
  $V_i \leftarrow (V_i - V_{i+1} * L_i) * D_i$

This algorithm is then implemented as follows `TDSMForwardBackward`, in the file *TDSMForwardBackward.hxx* shown in Figure 5:

```
1  void operator() (int i) const {
2    MView m=mcView_[i];
3    VView v=vView_[i];
4    const int size=
5      MView::getSize(m.getShape());
6
7    // temporary variable
8    // vec_1=v[i-1] (forward)
9    // vec_1=v[i+1] (Backward)
10   float vec_1;
11
12   // Forward Substitution
13   vec_1=v[0]*m(TDSMView::diag, 0);
14   v[0]=vec_1;
15   for (int j=1 ; j<size ; j++){
16     vec_1=(v[j]-(vec_1*m(TDSMView::lower,
17                                 j-1)))*
18           m(TDSMView::diag, j);
19     v[j]=vec_1;
20   }
21
22   // Backward substitution
23   vec_1=vec[size-1]*
24         m(TDSMView::diag, size-1);
25   v[bs-1]=vec_1;
26   for (int j=size-2 ; j>=0 ; j--){
27     vec_1=(v[j]-(vec_1*m(TDSMView::lower,
28                                 j)))*
29           m(TDSMView::diag, j);
30     v[j]=vec_1;
31   }
32 }
```

Processor	Type	# of cores	freq.	RAM	Compiler
Intel Xeon E5570 (Nehalem)	CPU	4	2.9GHz	18 GB	g++ 4.3.3
Intel Xeon E5410	CPU	4	2.3GHz	8 GB	g++ 4.1.3
AMD Opteron 2220	CPU	2	2.8GHz	8 GB	g++ 4.1.3
NVIDIA C870	GPU	128	1.3GHz	1.5 GB	nvcc 2.2
NVIDIA C1060	GPU	240	1.3GHz	4 GB	nvcc 2.2
NVIDIA FX5800	GPU	240	1.3GHz	4 GB	nvcc 2.2

Table 1: Configuration of our test machines

5. EXPERIMENTAL PERFORMANCES

5.1 Test Protocol

Table 1 lists the specifications of the machines and of the compilers used in the experiments. Note that all our test machines are dual-processors (2 sockets). The matrix have between 100 and 2×10^6 blocks on CPU and between 100 and 1×10^6 on GPU. Matrix blocks have a size of 100×100. All data are single precision floating point reals. The time measured is the time spent in the call of the `MTPS::Parallel_for` function. Curves are plotted with the same ordinate unit: number of blocks computed by second. On CPU, measures for differents numbers of threads have been made. The number of threads is indicated in the legend as indice (e.g. TBB_4 means that 4 threads are running). TBB_{-1} correspond the the default mode of TBB : one thread on each core.

5.2 Cholesky Decomposition

Figure 6(a) shows the performances of Cholesky decomposition on our CPUs. One can observe that performances increase linearly with the number of threads up to the number of cores. This tends to show that this operation is limited by the computation power of processors and not by the memory bandwidth. This is explained by the square root and the division operations which are very expensive on CPUs.

Figure 6(b) shows the performances of Cholesky decomposition on our GPUs. In this case, performances are limited by the memory bandwidth. Indeed, data throughput obtained (55.4 GB/s on the C870 and 74.7 GB/s on the other two cards) are very close to the data throughput observed using `cudaMemcpy` (respectively 62 GB/s and 76 GB/s). On GPUs, square root and division are implemented in hardware and thus are less expensive.

Finally, this operation performs 11.8 times faster on a C1060 GPU than on 2 E5570 quad-cores CPUs.

5.3 Forward and Backward Substitution

Figure 7(a) shows the performances of the forward and backward substitution on CPU. This operation requires 6 floating point operations and 8 memory accesses per element. However, on CPU, the cache memory effectively reduces the number of memory accesses down to 4 (3 reads during forward and 1 write during backward). Even so, the memory bandwidth required by the processors to get data from RAM is very high. This explains why performances saturate on the E5410 when more than 6 threads are launched. On the other hand, the memory architecture of the E5570 is more efficient and provides a higher bandwidth which avoids this limitation. On the Opteron 2220, this limitation does not appear because the limited parallelism (4 threads) does not require a memory bandwidth as high as on the two other CPUs.

Figure 7(b) shows the performances of the forward and backward substitution on GPU. There is no memory cache on GPU. Hence more memory accesses than floating point operations are required: this operation is obviously limited by the memory bandwidth. Once again, this is confirmed by the data throughput obtained: 54.8 GB/s on the C870 and 73.8 GB/s on the two other GPUs. Let us point out these two observations:

- the number of memory accesses required by this operation is twice the number of operations required by the decomposition,
- this operations computes half as many blocks per second as the decomposition

This confirms that for these two operations, elapsed time only depends on the number of memory accesses.

Finally, this operation performs (only) 2.4 times faster on a C1060 GPU than on 2 E5570 quad-cores CPUs.

6. RELATED WORKS

With the emergence of new parallel architectures, many works try to use C++ metaprograming to generate parallel applications. Our work is a C++ generic library at the junction between two approaches:

- utilization of parallel skeletons,
- data layout adaptation,

which are used to build optimized multi-target applications.

Generic libraries providing parallel skeletons exist for multicore CPUs and for GPUs. On CPU, Intel's TBB [18] and Microsoft's PPL [13] use containers and algorithms similar to those in the STL. On GPU, Thrust [21] and CUDPP [5] provide a small subset of the STL containers and algorithms. Although efforts are made to abstract the underlying architecture, users still need to be aware of this architecture to use it efficiently. In other words, a user still has to optimize his code for a given architecture that will be used to run the application.

Libraries allowing data parallelism on distributed memory systems (e.g. PC clusters) supply a more abstract data layout. For instance, STAPL [1] provides parallel containers as well as parallel iterators (called pRanges) [20]. HPC++ Parallel Standard Template Library (PSTL) [9], Charm++ [10] and DaTel [3] supply comparable functionalities. However, these libraries still require an awareness of the underlying architecture. Indeed, data are not distributed the same way on each machine and user has to deal with the distribution in order to achieve optimal performances.

MTPS is a design for multitarget programming. We identified the data layout as the most heterogeneous part between the implementation for two target architectures. Thus, our contribution is an abstract data layout that adapts the memory access pattern through parallel skeletons. Thanks to its limited scope, MTPS' can apply domain specific optimizations. Fitting one's data stucture into MTPS formalism

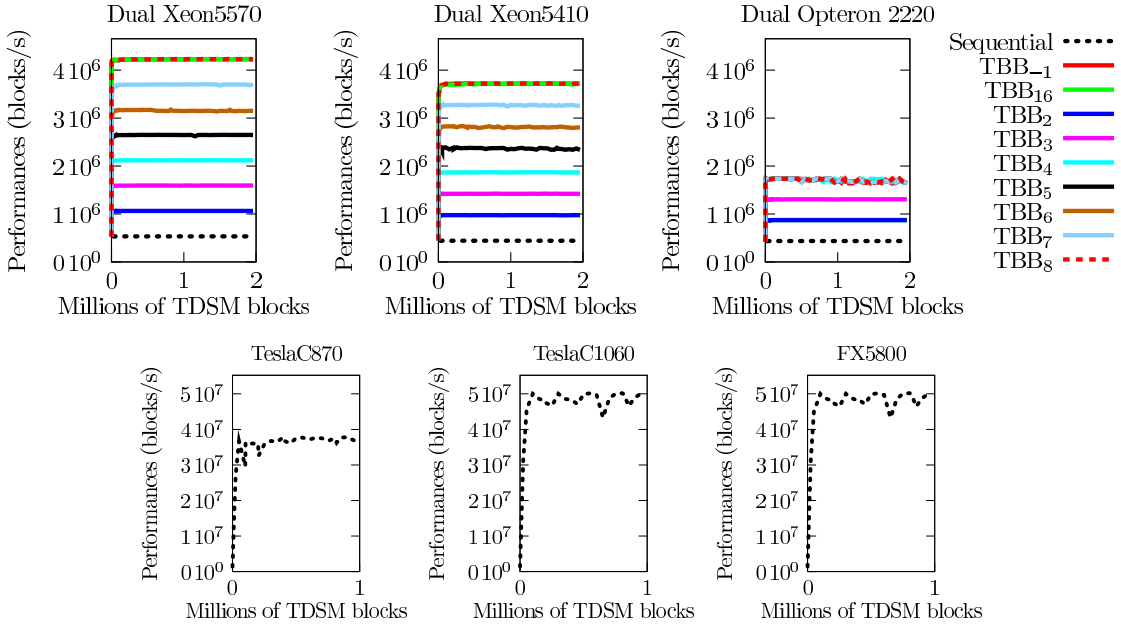

Figure 6: Performances of the Cholesky decomposition on CPU using Intel TBB (top) and on GPU NVIDIA CUDA (bottom).

allows one to write architecture-independent codes the performances of which are close to the hardware limits.

7. CONCLUSION AND PERSPECTIVES

In this paper, we have presented the design and the implementation of a new generic library that allow to compile and execute from the same source code both on CPU using Intel TBB and on GPU using NVIDIA CUDA. This implementation is based on two abstractions. First, a generic data structure which allow the specialization of the data storage for each target architecture. Second, parallel skeletons which allow efficient memory accesses. These abstraction enable hardware specific optimizations. Finally, performances obtained have been shown to be close to hardware limits.

For further developments, we consider two main directions. On the one hand, having more skeletons would allow to widen the scope of MTPS applications. On the other hand, having interfaces for other target architectures, including architecture with distributed memory like clusters would help us in validating our abstraction concerning the data layout. These two directions of research will be investigated in a close future.

8. ACKNOWLEDGMENT

Authors want to thank Region Lorraine and ANRT that have supported this research.

9. REFERENCES

[1] P. An, A. Jula, S. Rus, S. Saunders, T. G. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In H. G. Dietz, editor, *LCPC*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2001.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[3] H. Bischof, S. Gorlatch, and R. Leshchinskiy. Generic parallel programming using c++ templates and skeletons. *Domain-Specific Program Generation*, pages 107–126, 2004.

[4] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[5] cudpp: *http://gpgpu.org/developer/cudpp*.

[6] K. Czarnecki, J. T. Odonnell, J. Striegnitz, Walid, and Taha. Dsl implementation in metaocaml, template haskell, and c++. *LNCS: Domain-Specific Program Generation*, 3016(2):51–72, 2004.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.

[8] Intel. MKL web page can be found from: *http://www.intel.com*.

[9] E. Johnson and D. Gannon. Programming with the hpc++ parallel standard template library. In *In Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (PP'97*, 1997.

[10] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[11] W. Kirschenmann, L. Plagne, S. Ploix, A. Ponçot, and S. Vialle. Massively parallel solving of 3D simplified P_N equations on graphic processing units. In

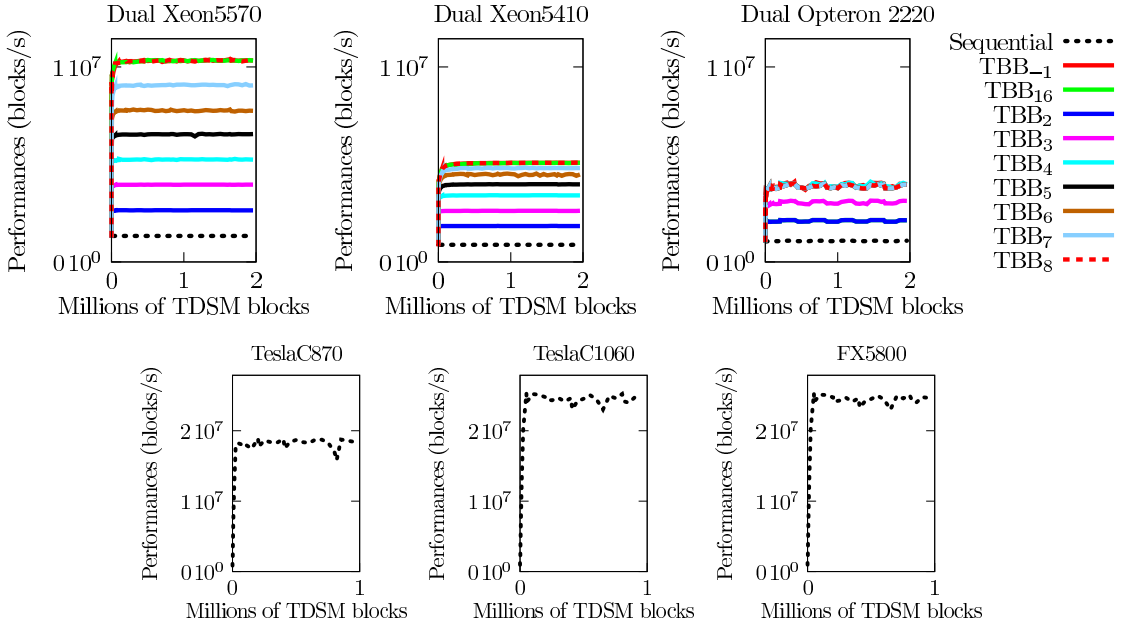

Figure 7: Performances of the forward and backward substitution on CPU using Intel TBB (top) and on GPU NVIDIA CUDA (bottom).

Proceedings of Mathematics, Computational Methods & Reactor Physics, May 2009.

[12] MTL4: *http://www.osl.iu.edu/research/mtl/mtl4/*.

[13] Microsoft. Microsoft PPL web page can be found from: *http://msdn.microsoft.com/en-us/concurrency/default.aspx*.

[14] Netlib. BLAS: *http://www.netlib.org/blas*.

[15] NVIDIA. *CUDA* Programming Guide *2.0*, July 2008.

[16] L. Plagne and A. Ponçot. Generic programming for deterministic neutron transport codes. In *Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*, Palais des Papes, Avignon, France, September 2005.

[17] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[18] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[20] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Associative parallel containers in stapl. In V. S. Adve, M. J. Garzarán, and P. Petersen, editors, *LCPC*, volume 5234 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.

[21] Thrust: *http://code.google.com/p/thrust/*.

[22] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[23] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–25, 2001.

[24] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.